Effektiv, effizient und sicher

Theorie und Praxis von Zufallszahlengeneratoren

Christian Heitzmann

Für kryptografische Aspekte – eine der Grundsäulen der IT-Sicherheit – ist es unabdingbar, dass ein Computer zufällige Zahlenwerte erzeugen kann. Allerdings ist ein Computer per definitionem eine rein deterministische Rechenmaschine, was dem Konzept des Zufalls elementar zu widersprechen scheint. Dieser Artikel beleuchtet die Hintergründe der Zufallszahlengenerierung und einige wichtige Kriterien für den Einsatz in der Praxis.

Bevor Sie, lieber Leser, mit dem Studium dieses Artikels fortfahren, sind Sie gebeten, das kurze Java-Programm aus Listing 1 auf Ihrem Rechner auszuführen. Es generiert mit der Ihnen sicherlich bekannten Klasse `java.util.Random` 10 mal 10 Integer-Werte im Wertebereich zwischen 100 (inklusive) und 1000 (exklusive), also insgesamt 100 dreistellige Zufallszahlen. In moderner Java 8-Notation wurden hierzu Streams verwendet, um den Quellcode knapp und die Ausgabe elegant zu gestalten.

```
import java.util.*;
import java.util.stream.*;

public final class IntroductionDemo {
  public static void main(String[] args) {
    Random random = new Random(10);
    for (int i = 1; i <= 10; i++) {
      System.out.println(random.ints(100, 1000).limit(10)
        .mapToObj(Integer::toString)
        .collect(Collectors.joining(", ")));
    }
  }
}
```

Listing 1: 100 persönliche Zufallszahlen

Die hellseherischen Fähigkeiten des Autors zusammen mit der hervorragenden Koordination von Verlag, Abonnenten-Service und Druckerei ermöglichten es, dass Sie hier und jetzt Ihre individuelle Vorhersage der auf Ihrer Konsole angezeigten Zufallszahlen in den Händen halten. Sie lauten:

```
613, 880, 493, 190, 446, 956, 497, 788, 481, 614
623, 399, 691, 808, 895, 580, 686, 353, 473, 638
693, 809, 295, 108, 735, 749, 474, 870, 108, 450
948, 702, 772, 396, 917, 875, 710, 110, 955, 833
639, 137, 792, 712, 368, 699, 898, 694, 741, 743
405, 861, 271, 195, 681, 789, 708, 592, 708, 868
360, 514, 636, 599, 536, 844, 290, 945, 399, 828
722, 380, 968, 237, 351, 163, 863, 481, 279, 624
133, 962, 508, 502, 788, 561, 279, 459, 748, 107
737, 957, 307, 707, 136, 439, 223, 548, 232, 939
```

Sie glauben es immer noch nicht? Dann generieren Sie 1000 Zeilen (ändern Sie den Zähler in der `for`-Schleife auf `i <= 1000`). Ihre persönlichen letzten drei Zufallszahlen lauten dann `878`, `288` und `546`. Da wird selbst Uri Geller neidisch!

Effektiv – Random und lineare Kongruenzgeneratoren

Die hoffentlich amüsante Einleitung bringt gleich zu Beginn eine entscheidende Schwachstelle von einfachen Zufallszahlengeneratoren auf den Punkt: *Vorhersagbarkeit*. Das Codebeispiel aus Listing 1 sollte mit den ganzen 10ern, 100ern und 1000ern davon ablenken, dass dem Zufallsgenerator in `Random` im Konstruktor ein sogenannter *Seed* (deutsch in etwa: Samen, Keim) mitgegeben wird, nämlich 10, der als interner Startwert fungiert. Beim Algorithmus, der in `Random` implementiert ist, handelt es sich um nichts anderes als eine etwas kompliziertere Rechenvorschrift, die mit jedem Methodenaufruf von `nextInt` (oder anderen) aus dem vorherigen Wert einen neuen Wert berechnet. Bulk-Methodenaufrufe, wie sie im Codebeispiel implizit in den Streams vorkommen, führen intern ebenfalls zu diesen einzelnen Methodenaufrufen.

Es handelt sich also um einen sogenannten *Algorithmus mit Gedächtnis*, der eine wohldefinierte *Sequenz* von Zahlen liefert. Dabei spielt es keine Rolle, ob 100 Zahlen oder 1 000 000 generiert werden. Ist der Seed bekannt, kann die gesamte Sequenz lückenlos rekonstruiert werden.

Jetzt wird mancher Leser vermutlich einwenden, er habe es noch nie erlebt, dass sich seine Applikationen, die eine Zufallskomponente enthalten, immer stets exakt gleich verhalten hät-

Christian Heitzmann ist Gründer und Geschäftsführer der SimplexaCode AG in Luzern, die sich auf Softwareentwicklung, -schulung und -beratung vor allem für MINT-Anwendungen und technische Implementierungsthemen in Java spezialisiert hat. Er ist seit 15 Jahren mit Java vertraut und hat während vieler Jahre Algorithmen und Mathematik unterrichtet.
E-Mail: christian.heitzmann@simplexacode.ch

ten. Weder spielt der MP3-Player immer die gleiche Zufallsreihenfolge der Lieder ab, noch entstehen beim Bildschirmschoner immer die gleichen Muster. Der Grund dafür ist ganz einfach: Wird `Random` im Konstruktor kein Wert mitgegeben (vermutlich der Standardfall), dann initialisiert sich der Zufallsgenerator selbst, und zwar anhand von `System.nanoTime`. In der Dokumentation des Standardkonstruktors heißt es dazu leicht überformell: *„This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor."* [JAPIRan], was beim Aufruf von `System.nanoTime` zweifelsohne der Fall ist.

Für die alltäglichen Anwendungen, wie den erwähnten MP3-Player oder den bunten Bildschirmschoner, gerne auch für Computerspiele, mag das völlig ausreichend sein. Entscheidend ist, dass die Zufallswerte nicht die gleichen wie gestern oder vor einer Stunde sind. Bei der Schlüsselgenerierung eines Online-Banking-Servers sieht die Sache aber schon anders aus. Das Wissen, der Banking-Server hätte für die Generierung der Zufallzahlen einen Generator mit `System.nanoTime` initialisiert, wäre fatal. Ein paar Sekundenbruchteile vor- und zurückspulen, mit einem Parallelrechner die verschiedenen Möglichkeiten durchspielen, und schon ist es möglich, die gesamte Zufallszahlensequenz – und damit auch die generierten Schlüssel – zu rekonstruieren.

Linearer Kongruenzgenerator und Periodenlänge

Im Kern besteht `Random` aus einem Algorithmus, der sich *linearer Kongruenzgenerator* (englisch: linear congruential generator, LCG) nennt, und im Prinzip verblüffend einfach ist. Die Methode geht auf die Anfänge des Computerzeitalters zurück und stammt von Derrick H. Lehmer (1905 – 1991). Bereits 1949 wurde diese sehr einfache Methode vorgestellt und ist die bis heute am häufigsten verwendete Art von Zufallszahlengeneratoren.

Grundprinzip ist eine einfache Rekursionsgleichung, bei der eine Zahl einer Zufallszahlenfolge mithilfe einer linearen Funktion und einer modularen Division (dem Rest einer ganzzahligen Division) aus der direkten Vorgängerzahl anhand folgender Formel bestimmt wird:

- $x_{i+1} = (a \cdot x_i + c) \bmod m$

m ist dabei der *Modul*, a ist der *Multiplikator* und c das *Inkrement*. Man nimmt also den vorherigen Wert, multipliziert ihn mit a, addiert eine Konstante c hinzu und nimmt dann den Rest, der aus der Division durch m entsteht. In Listing 2 ist ein solcher linearer Kongruenzgenerator mit den Werten $m = 16$, $a = 5$ und $c = 1$ einmal nachgebaut.

Wenn jeder Schritt durch den Rest einer Division durch (hier) 16 abgeschlossen wird, dann liegt es auf der Hand, dass sich eine solche Sequenz nach (hier) spätestens 16 Schritten wiederholen muss. War zum Beispiel der Wert irgendwann einmal 13, dann führte dies dazu, dass der darauffolgende Wert eine 2 wurde, denn $(5 \cdot 13 + 1) \bmod 16 = (65 + 1) \bmod 16 = 66 \bmod 16 = 2$, da $66 = 4 \cdot 16 + 2$, der Rest der Division durch 16 also 2 beträgt. Tritt also irgendwann wieder die 13 als Ergebnis auf, dann wird im darauffolgenden Schritt sicher wieder 2 folgen. Da eine Modulodivision durch

```
public final class LinearCongruentialGenerator {
  private static final int M = 16;
  private static final int A = 5;
  private static final int C = 1;
  private static int xi = (int) (System.nanoTime() % M);

  public static void main(String[] args) {
    for (int i = 1; i <= 10; i++) {
      System.out.format("%d\t%d\t%d%n",
        Integer.valueOf(nextInt()),
        Integer.valueOf(nextInt()),
        Integer.valueOf(nextInt()));
    }
  }
  public static int nextInt() {
    int xiplus1 = (A * xi + C) % M;
    xi = xiplus1;
    return xiplus1;
  }
}
```

Listing 2: Linearer Kongruenzgenerator

16 höchstens 16 verschiedene Endergebnisse haben kann (nämlich alle Zahlen zwischen 0 und 15), wird es höchstens 16 Schritte dauern, ehe ein Ergebnis erneut auftritt und sich der ganze Zyklus damit wiederholt.

In der Tat tritt das auch im Codebeispiel aus Listing 2 auf, auch wenn es aufgrund der Dreierblockformatierung nicht sofort ins Auge springt. Der Sequenzanfang wird aufgrund der Initialisierung mit `System.nanoTime` bei jedem Leser variieren, auf jeden Fall wird sich jedoch diese wiederholende Sequenz in der Ausgabe finden lassen: `5, 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, 4`

Die Bestimmung der Werte m, a und c ist eine Wissenschaft für sich, gehört grundsätzlich in die Hände von Mathematikern und wird in der Literatur intensiv behandelt [Knu98, Gen03, Pres07]. Die erste große Herausforderung besteht darin, eine Kombination der drei Werte m, a und c zu finden, die die größtmögliche Periodenlänge zur Folge hat, nämlich m. Dies ist alles andere als trivial. Die Parameterwerte $m = 100$, $a = 65$ und $c = 14$ resultieren zum Beispiel nur in einer Periodenlänge von 2 (obwohl man gerne 100 hätte), die Parameterwerte $m = 32\,768$, $a = 654$ und $c = 6841$ fahren sich sogar nach ein paar Schritten auf einen einzigen Wert fest (nämlich 15 395), obwohl man hier gerne 32 768 verschiedene Werte hätte. Der Leser kann dies anhand der ihm vorliegenden Musterimplementierung des LCG gerne nachvollziehen.

Spektraltest

Die zweite große Herausforderung besteht darin, trotz hervorragender Periodenlänge so etwas wie „Zufall" in den Sequenzen zu haben. Angenommen, die Parameter seien $m = 10\,000$, $a = 1$ und $c = 1$. Die resultierende Periodenlänge ist mit 10 000 zwar perfekt, die Zahlenfolge ..., 5022, 5023, 5024, 5025, 5026, ... ist aber alles andere als zufällig.

Hier setzt der sogenannte *Spektraltest* an. Er betrachtet immer Zahlenpaare aus einem Zufallszahlengenerator. Die erste Zahl des Zahlenpaares entspricht der x-, die zweite Zahl der y-Koordinate eines Punktes in einem Koordinatensystem. Die so resultierende Punktemenge sollte schön zufällig und gleichmäßig aussehen (so wie eine frisch eingeschneite, unbefahrene Straße). Interessanterweise scheitern aber alle linearen Kongruenzgeneratoren an diesem Test. Abbildung 1 zeigt den Spektraltest für die Parameter $m = 32\,768$, $a = 3423$ und $c = 1$. Die Periodenlänge beträgt 2048, und alle Punkte befinden sich auf einem geneigten Raster. Wahrlich kein guter Zufall.

Die Parameter $m = 32\,768$, $a = 3421$ und $c = 1$ sehen etwas vielversprechender aus. Die Periodenlänge beträgt volle 32 768. Be-

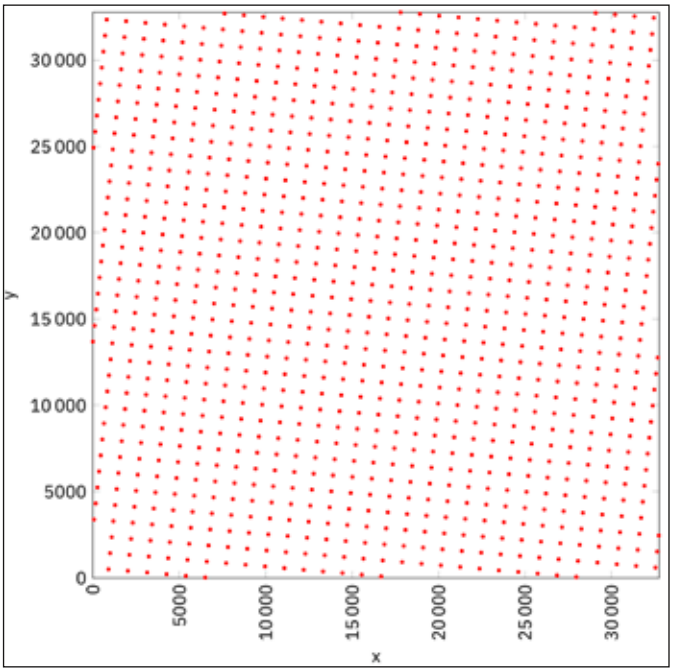

Abb. 1: 2D-Spektraltest für m = 32 768, a = 3423, c = 1, n = 1200

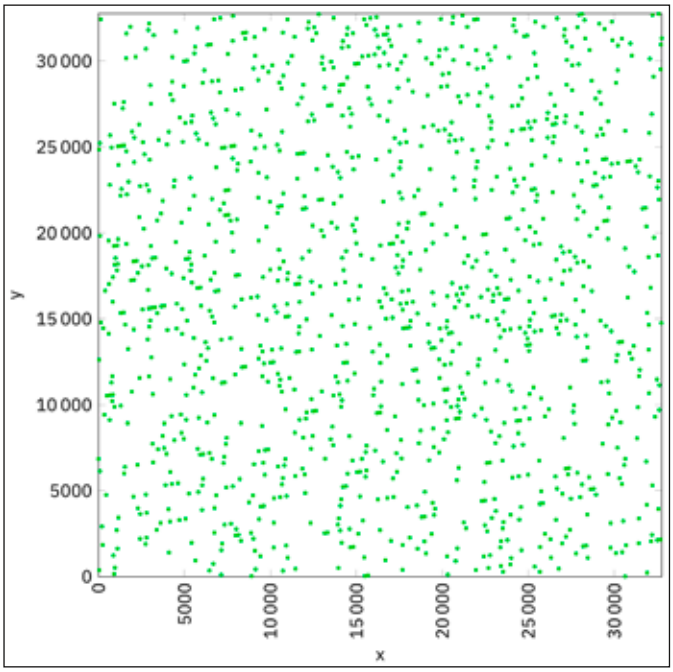

Abb. 2: 2D-Spektraltest für m = 32 768, a = 3421, c = 1, n = 1200

trachtet man die ersten 1200 Wertepaare, ergibt sich Abbildung 2. „Schießt" man weiter auf die Fläche, so ergibt sich bei 12 000 Wertepaaren ein Bild wie in Abbildung 3. Hier ist jetzt wieder ein ganz klares Muster zu erkennen, denn alle Punkte liegen nur auf den schrägen Linien.

Versagt der Zufallsgenerator nicht bereits offensichtlich beim 2D-Spektraltest, dann wird er es beim 3D- oder einem noch höherdimensionalen Spektraltest tun. Abbildung 4 zeigt einen 3D-Spektraltest für die Parameter $m = 32\,768$, $a = 3423$ und $c = 1$, in dem jeweils Punktetripel herangezogen werden: Der erste Wert entspricht der *x*-, der zweite Wert der *y*-, und der dritte der *z*-Koordinate eines Punktes im 3D-Koordinatensystem. Kommen – wie in Abbildung 4 – die Punkte auf Ebenen zu liegen, so ist auch dies ein schlechtes Indiz für die Zufälligkeit der Zahlen.

Stirb langsam

Für Zufallszahlengeneratoren gibt es eine ganze Palette an (teilweise sehr amüsant klingenden) Gütetests. Die *Diehard*- [Diehard] und *Dieharder*-Testsuites [Dieharder] sind vermutlich die bekanntesten und bestehen aus einer Fülle von Tests, wie zum Beispiel dem „Birthday Spacings Test", dem „Parking Lot Test", „Monkey Test" usw.

Und wie verhält es sich mit dem LCG aus `Random`? Der darin implementierte LCG arbeitet intern mit 48 Bits, verwendet davon aber nur 32 Bits für das Ergebnis. Die Periodenlänge beträgt 2^{48}, was etwa 281 Billionen (eine 15-stellige Zahl) entspricht. Der Rückgabewert an den Methodenaufrufer ist jeweils ein „gekapptes" Ergebnis. Wer also `nextInt(100)` aufruft, erhält zwar nur Zahlenwerte, die kleiner als 100 sind, was aber nicht heißt, dass die Periodenlänge dann auch nur 100 wäre. Aufgrund der großen internen Zahlenmenge dürften im Alltag auch die Spektraltests zumindest für das menschliche Auge zu bestehen sein.

Lineare Kongruenzgeneratoren sind schlecht, sogar grottenschlecht. In der Literatur werden sie zerrissen, und es ist unverständlich, wieso sie – trotz ihrer nachweislich schlechten Eigenschaften – immer wieder Bestandteil von (neuen) Bibliotheken sind. Praktisch jede Programmiersprache, die mit einer Standardbibliothek ausgeliefert wird und eine Funktion für Zufallszahlen anbietet, hat in ihr eine mehr oder weniger direkte Form des LCG eingebaut. Die englische Wikipedia-Seite gibt einen Eindruck über seine Verbreitung und die in den jeweiligen Bibliotheken verwendeten Konstanten [WikiLCG].

Javas `Random` ist unter all den LCGs wohl noch der beste. Für die meisten alltäglichen Anwendungen dürfte es ausreichend sein, die nächste Zufallszahl nicht „mit vertretbarem Aufwand" vorhersagen zu können. Kann der Leser innerhalb der nächsten 5 Sekunden das nächste Zufallslied aus seiner Playlist vorhersagen? Nein. Kann er es vorhersagen, selbst wenn er eine Nacht darüber schläft? Nein. Kann er die Farbe des nächsten Bildschirmschonerstrichs vorhersagen, selbst wenn er soviel nachdenken darf, wie er möchte? Nein. In diesem Sinne ist der LCG ausreichend. Er erfüllt seinen Zweck, trotz theoretischem Determinismus den Anschein einer Zufälligkeit zu erwecken, der für nicht sicherheitskritische und nicht wissenschaftliche Anwendungen ausreichen dürfte.

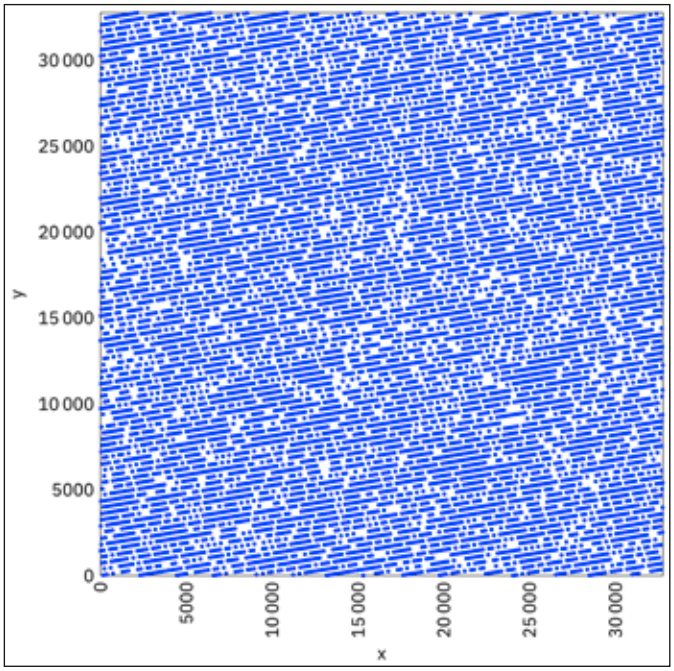

Abb. 3: 2D-Spektraltest für m = 32 768, a = 3421, c = 1, n = 12 000

Effizient – ThreadLocalRandom und SplittableRandom

Javas Standard-Zufallszahlengenerator `Random` ist threadsicher, das heißt, es können beliebig viele Threads gleichzeitig auf ein und dieselbe `Random`-Instanz zugreifen. `Random` kümmert sich intern um die notwendige Synchronisierung. Wie so oft bei nebenläufigen Methodenaufrufen ist es dabei nicht die eigentliche Synchronisierung, die teuer ist, sondern das Aufstauen von wartenden Threads als Folge der Synchronisierung. Oder anders ausgedrückt: Ein paralleler Programmfluss wird an solchen Stellen zu einem seriellen Flaschenhals.

Mit der Klasse `java.util.concurrent.ThreadLocalRandom` gibt es seit Java-Version 7 einen weiteren Zufallszahlengenerator, der mit einigen entscheidenden Verbesserungen aufwartet. Die schlechte Nachricht vorab: Auch er ist „nur" ein sogenannter *Pseudozufallszahlengenerator*, das heißt, alle erzeugten Zufallszahlenwerte sind – wie oben ausführlich erläutert – nicht wirklich zufällig, sondern rein deterministisch.

Hingegen kam inzwischen die Botschaft bei den JDK-Entwicklern an, dass man auf keinen Fall mehr auf den mangelhaften linearen Kongruenzgenerator setzen darf. Die mathematischen theoretischen Interna des „neuen" Zufallszahlengenerators gehen auf mindestens drei akademische Veröffentlichungen neueren Datums (2010 und aufwärts) und namhafte Autoren zurück. In einem Blog-Eintrag hat sich ein Softwareentwickler die Mühe gemacht, den Hintergründen des ausgeklügelten Zufallszahlengenerators auf die Spur zu gehen [Videla]. Die Theorie hier nun aufzuzeigen, würde den ganzen Artikel sprengen. Zusammengefasst kann festgehalten werden, dass der Algorithmus auf dem Stand der Technik

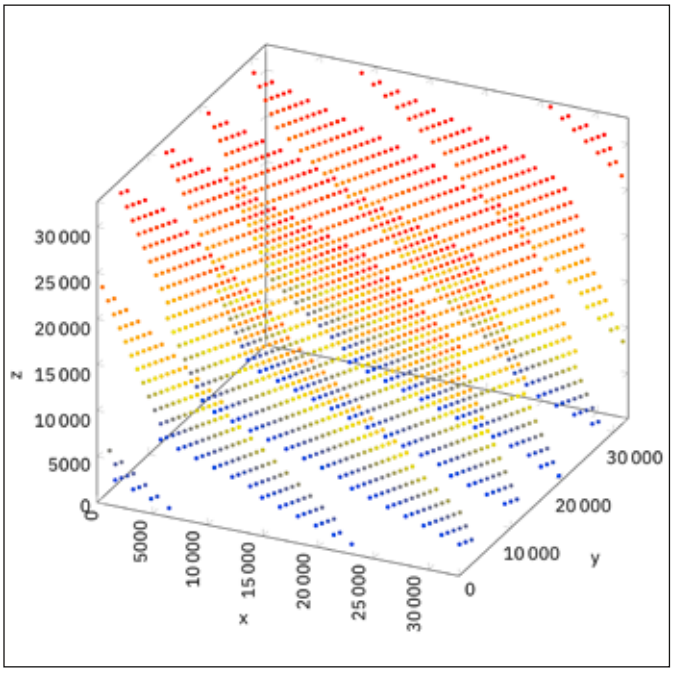

Abb. 4: 3D-Spektraltest für m = 32 768, a = 3423, c = 1, n = 12 000

ist, die gängigen statistischen Tests besteht und die Periodenlänge mindestens 2^{64} (18 Trillionen, eine 20-stellige Zahl) beträgt.

Wie der Name bereits impliziert, wurde `ThreadLocalRandom` für den Multithreading-Einsatz entworfen. Parallele Zugriffe werden aber nicht serialisiert, sondern im Gegenteil, jedem Thread steht automatisch eine intern eigenständige Instanz des Generators zur Verfügung. Der Aufruf hat daher immer nach dem Muster `ThreadLocalRan-`

dom.current().nextInt(...) (oder nextLong(...) oder nextDouble(...) usw.) zu erfolgen. Ein Aufstauen der Threads ist nun nicht mehr möglich.

Selbst wenn man keinen Einsatz mit mehreren Threads plant, ist ThreadLocalRandom wesentlich eleganter in der Anwendung. Während Random lediglich für nextInt(int upperBoundExclusive), nicht jedoch für die anderen primitiven Datentypen die Spezifizierung einer Obergrenze erlaubt, ist dies bei ThreadLocalRandom möglich. Für ints, longs und doubles werden die nextXXX-Methoden jeweils in drei Varianten angeboten (aufgezeigt am Beispiel long):

- nextLong()
- nextLong(long lowerBoundInclusive)
- nextLong(long lowerBoundInclusive, long upperBoundExclusive)

Auch der Aufruf des Zufallszahlengenerators ist aufgrund des immer gleichen statischen Aufrufmusters ThreadLocalRandom.current().nextXXX eleganter. Der ThreadLocalRandom ist da, wenn man ihn braucht, und stört nicht, wenn man ihn nicht braucht. Eine Initialisierung in einem Klassenattribut kann entfallen, ebenso wie der Aufruf der in der Praxis eigentlich untauglichen statischen Math.random()-Methode.

Seit Java 8 gibt es mit der Klasse java.util.SplittableRandom einen weiteren Pseudozufallszahlengenerator, der mathematisch ähnlich zu ThreadLocalRandom ist. Unter der Haube hängen die beiden Generatoren zusammen, insofern lässt sich ihre Güte als gleichwertig ansehen. SplittableRandom wurde primär für den Einsatz in *Fork/Join-Frameworks* entworfen, in denen sich große Probleme gemäß einem Teile-und-herrsche-Ansatz in zwei kleinere Teilprobleme zerlegen lassen, die dann parallel abgearbeitet (und vermutlich weiter unterteilt) werden. Um diesen Teilproblemen jeweils auch voneinander unabhängige Zufallszahlengeneratoren an die Hand zu geben, wurde SplittableRandom geschaffen. Eine ausführliche Abhandlung des Fork/Join-Frameworks findet der Leser in einem Blog-Eintrag des Autors [ForkJoin].

Der Einsatz von ThreadLocalRandom und SplittableRandom ist algorithmisch klar überlegen, in der Handhabung wesentlich eleganter und in Sachen Performanz unangefochten, vor allem im nebenläufigen Einsatz. *Selbst für einfache Anwendungen ist* ThreadLocalRandom *vorzuziehen. Es gibt für* Random *aus Sicht des Autors heute keine Einsatzberechtigung mehr.*

Sicher – SecureRandom

Für sicherheitskritische Anwendungen sind *echte* Zufallszahlengeneratoren unabdingbar. Mit der Klasse java.security.SecureRandom steht dem Entwickler ein solcher Nicht-Pseudozufallszahlengenerator zur Verfügung. Moment mal, wurde bis jetzt nicht eingehend durchgekaut, dass diese Generatoren in Wirklichkeit alle vorhersagbar sind? Richtig, aber SecureRandom ist anders als die anderen.

Für „echten" Zufall bedarf es einer *Entropiequelle*, also einer Quelle, die die „natürliche Unordnung" erfassen und daraus zufällige Bits erzeugen kann. Das Paradebeispiel für echt zufällige Prozesse ist der radioaktive Teilchenzerfall, allerdings nicht für den Einsatz innerhalb des heimischen Computers vertretbar. Thermisches, elektrisches oder akustisches Rauschen ist ebenfalls natürlich entropisch und lässt sich sogar mit einfacher Hardware implementieren. Man muss zum Beispiel nur seinen Mikrofoneingang bei Stille oder seine Webcam bei Dunkelheit aufdrehen, um genügend Rauschquellen einlesen zu können. Insbesondere die niederwertigsten Bits eines solchen Signals sind nicht vorhersagbar und damit echt zufällig.

Heutige Betriebssysteme sammeln permanent Entropiedaten ein. Dies sind zum Beispiel die Zeitabstände zwischen Tastaturanschlägen, Mausbewegungen, Systemzeiten, Scheduling-Verhalten, Netzwerkverkehr und dessen Kollisionen, teilweise eigene Hardwarekomponenten usw. In unixoiden Betriebssystemen stehen die daraus erzeugten Zufallsdaten als Stream unter /dev/random oder /dev/urandom zur Verfügung. Javas SecureRandom hängt sich – je nach gewähltem Algorithmus – an eine solche Entropiequelle und kann so nach einigen kryptografischen Zwischenschritten nicht-deterministische Zufallsbits oder -bytes zurückgeben.

```
import java.security.*;

public class SecureRandomAlgorithms {
  public static void main(String[] args) {
    for (Provider provider : Security.getProviders()) {
      for (Provider.Service service : provider.getServices()) {
        if (service.getType().equals("SecureRandom")) {
          System.out.println(service);
        }
      }
    }
  }
}
```

Listing 3: Liste aller SecureRandom-Algorithmen

Das Programm in Listing 3 gibt eine Liste aller auf dem eigenen System zur Verfügung stehenden SecureRandom-Algorithmen zurück. Auf dem MacBook Pro des Autors sind dies zum Beispiel:

- SecureRandom.NativePRNG
- SecureRandom.SHA1PRNG
- SecureRandom.NativePRNGBlocking
- SecureRandom.NativePRNGNonBlocking

Der Algorithmus NativePRNG sollte eigentlich immer eine gute Wahl sein, da er an die Entropiequelle des Betriebssystems anknüpft. Die Meinungen über SHA1PRNG gehen auseinander, bemängelt wird insbesondere die knappe bis nicht existente Dokumentation.

Ein SecureRandom-Objekt kann entweder via Standardkonstruktor erzeugt werden (SecureRandom random = new SecureRandom()), oder via getInstance(String algorithm)-Fabrikmethode (zum Beispiel SecureRandom random = SecureRandom.getInstance("NativePRNG")). Der Standardkonstruktor nimmt den ersten Algorithmus aus der Liste der registrierten Security-Providers, der SecureRandom unterstützt. Für die ersten Gehversuche sollte dieses Standardverhalten ausreichen.

Performanz und Fazit

Ursprünglich war geplant, diesen Artikel mit einem Performanz-Vergleich der verschiedenen Zufallszahlengeneratoren abzuschließen, doch nach einer kurzen Reflexion hat sich das als wenig sinnvoll herausgestellt. *Der Entscheid, ob ein kryptografisch sicherer Zufallszahlengenerator verwendet werden soll, darf nicht von Performanz abhängen.* Wird Sicherheit benötigt, gibt es darüber nichts zu diskutieren.

Ein auf den ersten Blick so unscheinbares Thema wie die Erzeugung von Zufallszahlen, dem die meisten Leser vermutlich noch nie Beachtung geschenkt haben, stellt sich bei näherer Betrachtung als eine grundlegende Problematik mit weitreichenden Folgen heraus. Der Autor hat sich bereits vor vielen Jahren in einer über 100-seitigen Seminararbeit mit der Erforschung von Zufallszahlen im Informatikunterricht beschäftigt (und stellt sie auf persönliche Anfrage gerne zur Verfügung).

Der „normale" Java-Programmierer darf als Kernbotschaft mitnehmen, dass Random vollumfänglich durch ThreadLocalRandom ersetzt werden kann und soll. Jedem Programmierer und Entscheidungsträger, der mit sicherheitskritischen oder wissenschaftlichen Anwendungen in Berührung kommt, muss die grundlegende Problematik

der Vorhersagbarkeit aller Pseudozufallszahlengeneratoren bewusst sein. Geht es an die Details, ist der Beizug eines Experten unausweichlich. Wie der heute überall gesuchte (aber nirgendwo gefundene) „Full-Stack-Developer", der am Montag an einer der vielen kurzlebigen JavaScript-Bibliotheken, am Dienstag am Backend und am Mittwoch an der Middleware-Business-Logik herumschraubt, in seinem „agilen Umfeld" überhaupt die Umgebungsbedingungen, die Zeit, das theoretische Hintergrundwissen und die Expertise für grundlegend durchdachte Sicherheitskonzepte vorfinden oder aufbringen soll, kann als rhetorische Frage so stehen bleiben.

Der „Java Cryptography Architecture Reference Guide" [JCA] bringt es auf den Punkt: *„You should always understand what you are doing and why: DO NOT simply copy random code and expect it to fully solve your usage scenario. Many applications have been deployed that contain significant security or performance problems because the wrong tool or algorithm was selected."* Dem ist nichts mehr hinzuzufügen.

Literatur und Links

[Code] Quellcode zum Herunterladen, https://link.simplexacode.ch/sv8b

[Diehard] G. Marsaglia, The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness, 1995, https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/

[Dieharder] R. G. Brown, Dieharder: A Random Number Test Suite, http://webhome.phy.duke.edu/~rgb/General/dieharder.php

[ForkJoin] SimplexaCode, The Fork/Join Framework, https://link.simplexacode.ch/529z

[Gen03] J. E. Gentle, Random Number Generation and Monte Carlo Methods, 2nd Edition, Springer-Verlag, 2003

[JAPIRan] Java Platform, Standard Edition 8 API Specification, Class Random, https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

[JAPISecR] Java Platform, Standard Edition 8 API Specification, Class SecureRandom, https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html

[JAPISplR] Java Platform, Standard Edition 8 API Specification, Class SplittableRandom, https://docs.oracle.com/javase/8/docs/api/java/util/SplittableRandom.html

[JAPITLR] Java Platform, Standard Edition 8 API Specification, Class ThreadLocalRandom, https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadLocalRandom.html

[JCA] Java Cryptography Architecture (JCA) Reference Guide, https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html#Introduction

[Knu98] D. E. Knuth, The Art of Computer Programming – Volume 2: Seminumerical Algorithms, 3rd Edition, Addison-Wesley, 1998

[Pres07] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery, Numerical Recipes – The Art of Scientific Computing, 3rd Edition, Cambridge University Press, 2007

[Videla] A. Videla, Java ThreadLocalRandom Explained, 2016, http://alvaro-videla.com/2016/10/inside-java-s-threadlocalrandom.html

[WikiLCG] Wikipedia, Linear congruential generator, https://en.wikipedia.org/wiki/Linear_congruential_generator#Parameters_in_common_use